unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-84-022	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A Formal Method For Specifying Computer Resources In An Implementation Independent Manner		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Daniel L. Davis		8. CONTRACT OR GRANT NUMBER*(s)*
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N: RR014-08-01 N0001483WR30346
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, Virginia 22217		12. REPORT DATE November 1984 13. NUMBER OF PAGES 37
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*
Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This paper is an investigation of a methodology for the formal specification of computer software or hardware resource interfaces. The objective of the methodology is to make possible the specification of implementation independent, and thus portable, interfaces for the development of software. This paper is concerned with the theoretical and conceptual issues of such a specification methodology, and for the most part is an adaptation of the methods of algebraic specification of data types to the specification of computer resources. This paper is the basis for a practical specification in progress.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

(2)

NPS52 84-022

NAVAL POSTGRADUATE SCHOOL

Monterey, California

A FORMAL METHOD FOR SPECIFYING COMPUTER RESOURCES IN AN IMPLEMENTATION INDEPENDENT MANNER

Daniel Davis

November 1984

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Commodore R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported by Contract from the Office of Naval Research.

This report was prepared by:

Daniel Davis

DANIEL L. DAVIS
Associate Professor of Computer Science

Reviewed by:

BRUCE J. MACLENNAN
Acting Chairman
Department of Computer Science

Released by:

KNEALE T. MARSHALL
Dean of Information and
Policy Science

A FORMAL METHOD FOR SPECIFYING COMPUTER RESOURCES IN AN IMPLEMENTATION INDEPENDENT MANNER

Daniel Davis

Department of Computer Science

Naval Postgraduate School

Monterey, California

Introduction

The purpose of this report is to describe a methodology for the specificaton and development of portable software environments for limited resource machines. The ultimate goal of this work is to be able to build a portable software development environment for development on and for limited resource machines. A limited resource development system is a single user system that includes a processor, memory, disk storage, display, keyboard, and list device. The target machine for development might be a dedicated processor system such as in a smart device such as a robot, or process control device, or it may be an application on the development system.

The specific problem addressed here is the development of a methodology for specifying resources, both physical resources and problem solving (software), in an implementation independent manner. This methodology can then be used to specify successive layers of resource abstraction, beginning with the physical resources at the lowest level and ending with problem solving abstractions at the highest level. To achieve this goal we need a conceptual framework that has the following features.

It must be presentable in a clear and precise form.

It must provide a complete and rigorous theory of abstract specification

It must include a practical theory of implementation

A number of people have worked on the related problem of specifications of abstract data types. The focus of this report is the application of similar techniques to the specification of physical resources.

To further clarify my objectives here, I wish to briefly describe some of the historical background leading up to the current work.

Traditionally, software environments have developed around a body of hardware and to a certain extent reflect this heritage. Such systems of software tend to develop into 'closed systems' of software, with very little possibility of movement between systems. Even though high level abstractions provided by high level languages provide some measure of software standardization and portability, they tend to create closed systems at a very high level, and thus their portability is limited because to port such a system, all the layers of software below them must be ported. Moreover, because of the problems in specifying the semantics of high level constructs of different languages through any consistent theory, it has proven difficult to translate programs in one language to another. These factors and the labor intensive nature of systems software development have combined to create closed systems.

The problem of creating portable software achieved greater significance when our ability to design and create new processors accelerated. Traditionally, only a few companies have existed to produce the hardware environments around which software has evolved. With the development of microprocessors and the microcomputer industry, the number of companies producing computers proliferated, and at least initially, no one computer manufacturer predominated. At the same time these small companies did not have the resources to develop an extensive body of software for their particular environments, particularly things such as high level language compilers. Thus a new set of conditions surrounding the design and development of software occurred. The result has been that standardization and abstraction occurred at levels above the hardware, but below the high level languages. Examples of this are CP/M, the P-system, and 'C' and the 'C' runtime system. These systems are a software abstraction of physical resources. From the historical perspective, this is one of the more interesting concepts that has arisen from the development of microcomputers.

For some time, it has been recognized that the operating system represents an abstraction of the hardware system that supports the layers of software built over it, that is, an operationg system is an abstraction of the physical resources of a system. Traditionally the operating system provides a standardized programmatic interface to the secondary memory resources, primary memory resources, processsors, and i/o resources.The most recent personal computing systems go a step beyond the traditional operating system by including more sophisticated abstractions of the console display.

The problems that must be faced in trying to specify the properties of a real or abstract physical resource are similar to the problems faced by linguists who try to specify the semantics of language constructs. It is

difficult to develop abstract models that are precise, capture the essential features of something real, yet do not obscure and complicate our ability to work with what is real. On the other hand, if we are able to successfully capture the essential features of something we know intuitively, the abstract model can become a tool that enables us to sharpen our intuition, and increase our understanding. Unless we can develop abstract models that allow us to clothe our intuitive notions with precision, we will remain at an impasse in our ability to know which of these ideas are important and which are not.

In the following section we will outline the main elements of a conceptual system developed for this purpose. In the later sections, this conceptual system is made precise and illustrated in some detail.

1.0

Conceptual Tools

There are a number of features of the problem of abstract specification that naturally lead us to draw on mathematical discipline. The methodology we use must be representation independent. The methodolgy must give us a method of proving the correctness of our assertions about formal specifications and their implementations. We must be able to combine and compose specifications. The methodology we use should encourage a discipline of care and precision. At the same time we should attempt to avoid unnecessary abstraction or concepts that do not directly improve the correct use of the methodology.

Most of the concepts that we use here were developed to specifiy the semantics of high level language constructs, particularly, the specification of abstract data types. Since the specification of a portable programmatic interface has its origins in this work, and since these concepts are more readily understood in its early form, we will begin with an informal treatment of abstract data types.

1.1

Abstract Data Types

In their most common usages, abstract data types are simply problem solving resources. Some astract data types, for example a stack, are also abstractions of physical resources. Our purpose is to develop a theory of specification that can be used to describe either problem solving (software) resources or physical resources. To do this, we use a theory of abstract data types that has been developing over a number of years, and has involved a number of different researchers. The primary references to this work can be found in Goguen [1978] and Guttag [1978].

One of the simplest and most common data types in mathematics and computer science is boolean. We will use this data type to introduce our general methods.

Note first that a data type consists of more than the values of the type. The type is a composite of the values and the operators used with the type. In traditional usage, the set of values denotes the data type, when in fact, the aggregate of operations and values denotes the type. There is a similar misconceotion of the function concept in mathematics. Often a function is denoted by just its rule, when in fact it is an aggregate of domain, codomain, and rule.

For the boolean type, there are two values used, which may be denoted by T and F and several fundamental operators such as '¬' (logical negation), '&' (logical conjunction) and '|' (logical disjunction). Finally, there are relations that must hold for these operators as given by the traditional truth tables:

X	¬(X)
T	F
F	T

X	Y	&(X,Y)
T	T	T
T	F	F
F	T	F
F	F	F

X	Y	|(X,Y)
T	T	T
T	F	T
F	T	T
F	F	F

With the above definitions we are able to establish the truth of other relations:

The idempotent law for negation

The associative law for conjunction

The commutative law for conjunction

The distributive law for conjunction and disjunction

The DeMorgan laws

Obviously, there are other realizations of the data type that we normally call the boolean type. The symbols used for the data values may be {0,1}, the operators may be given different notations, etc. The fundamental operators may also be different. For example, the fundamental operators may be negation and implication. It is generally understood that this set of operators defines the 'same type'. Also, it is clear that this data type admits many other operators, exclusive disjunction, for example. It is clearly difficult to capture the essence of a type itself, independently of a particular realization of it. This is one of the problems that a theory of abstract specification must solve.

The things that are useful about a data type are not just the values of the type and the operators of the type, but the expressions we can build from values and operators. We use expressions to calculate with boolean values, so we need to 'evaluate' expressions and to determine if two expressions are 'equal', etc. Expressions are built from values and operators by abiding by the domain constraints of the operators, and using composition of operators. For example, all the following are obviously correctly formed expressions, assuming a prefix form for the operators.

¬(¬(&(T,F))

&(T,¬(&(F,¬(T)))

We also form expressions with 'free variables':

!(¬(x),&(T,y))

where of course x and y are the 'free' variables. Often we want to determine if two expressions are 'formally equal'. In particular we have reason to believe that every expression without free variables is equal to either T or F. Or we may have reason to believe that we can 'prove' that the expression ¬(&(x,y)) equals !(¬(x),¬(y)). In general, whenever we create and use a 'data type', we are potentially interested in the set of all expressions involving values of the type or free variables on the type. These objects are the abstract representatives of the things in the real world modeled by the data type. In fact, Hoffman and O'Donnell [1982] have recently expressed the view that much of computing involves no more than the transformation of complex expressions to recognizable form.

1.2

Algebras

The aggregate made up of specific sets of values, operators, and expressions form what is called an 'algebra'. Basically an algebra is a composite structure consisting of operations and sets. The sets describe the types of operands and results. The operations define all the ways that results are determined from operands. In the general case, the operations can have multiple operands of mixed type. The types of the operands are called 'sorts'. Boolean is a sort of the boolean data type. Operators may have multiple operands of mixed sort and give a result of a fixed sort. An operator is simply an n-ary function of the form:

op: A1,A2,A3,...,An -> A

where A1,...,An,A are carrier sets of sort S1,...,Sn,S respectively. The distinction between the 'type' of a set and its name is intentional.

In our description of an algebra, the operations are assumed to be explicitly defined functions on explicitly defined sets. If, however, we intend to use these concepts for the specification of real objects, we must be careful to avoid the specification of operations or sets that are not constructible by finitary methods. For example, the set of real numbers is not constructible by finitary methods. Also many of the operations used in mathematics assume non-finitary principles in their construction. Thus it is important to use care in the choice of which principles we

assume to construct the objects we use to represent the real objects we are attempting to specify.

We must also be sure that the method of specification itself has no representational bias. In the example above, we do not want to say that the boolean data type consists of the operators above on the sets above, since there are other operations and sets that represent this type equally well.

Similarly, we do not wish to specify a resource in a computing system as consisting of a specific processor, memory, disk, etc. but by the abstract functional properties these objects provide. However, we also have to account for the situation in which two systems which appear functionally different, are in fact functionally equivalent.

1.3

Algebraic Specifications

The manner in which algebraic specifications solve these problems is by first specifying 'templates' for the sets and operations that are being specified, and 'axioms' for the properties that any actual sets and operations must satisfy to meet the specification. The templates do not make any assumptions about the elements of the sets or the way that the operations act on elements. All such properties are specified in the axioms. Then rules are given to determine when two 'template' specifications specify the same thing. The ideas can be illustrated with the boolean data type.

First we require that there be exactly one set whose 'type' will be described by the name 'Bool'. There must be two 'constants' (0-ary operations) of type Bool, named 'True' and 'False'. Then there must be exactly two operations, one unary and one binary, with names, 'Not' and 'And', with appropriate functional type. Summarizing,

```
True: -> Bool
False: -> Bool
Not: Bool -> Bool
And: Bool,Bool -> Bool
```

Next, the following 'axioms' must hold:

```
Not(True) = False
Not(Not(x)) = x
And(True,x) = x
And(False,x) = False
And(x,y) = And(y,x)
And(And(x,y),z) = And(x,And(y,z))
```

The axioms above were chosen to compactly describe what are claimed to be all the essential properties of the operators. Note that nowhere is there a specification of the number of elements in any set that plays the role of 'Bool'. There are constant operations 'True' and 'False' whose values must be in the set playing the role of Bool, but there is no guarantee that these values are distinct, or that they are the only values.

The above specification can be codified into a compact syntax:

```
SPECIFICATION Boolean

        SORTS
                Bool
        OPS
                True: -> Bool
                False: -> Bool
                Not: Bool -> Bool
                And: Bool,Bool -> Bool
        AXIOMS
                Not(True) = False
                Not(Not(x) = x
                And(True,x) = x
                And(False,x) = False
                And(x,y) = And(y,x)
                And(And(x,y),z) = And(x,And(y,z))
```

Algebraic specifications always occur in two parts. The first part includes the sorts and the ops and is called the signature. The second part consists of the axioms. The axioms above are described as equations between terms with free variables. Axioms may also be 'conditional equations'. A equation is conditional if it has the form:

E1,E2,...,En => E

where E1,E2,...,En, and E are equations between terms.

We say that an algebra has the same signature as a specification if there is a one to one correspondence between the sorts and operations of the specification and the carrier sets and the operations of the algebra that is consistent with the type properties of the operations in the specification.

If 'Bool' is associated to the set {T,F},'True'is associated to the constant function whose value is T, 'False' is associated to the constant function whose value is F, 'Not' is associated to the unary operator ¬, and 'And' is associated to &, then it follows that the algebra we have discussed previously satisfies the above specification. Also this algebra clearly satisfies the axioms. Note

however there are many other algebras that also satisfy the above specification. For example, associate to the sort 'Bool' the set {a}. Associate to the 0-ary operators 'True' and 'False', the constant function on {a} whose value is 'a'. Associate to 'Not' the trivial unary function on {a} that is the identity. Associate to 'And' the trivial binary function on {a}. This algebra has the correct signature and it is not difficult to show that it satisfies the axioms. Yet we would not say this second algebra is representative of the 'Boolean' type. Thus there is a clear distinction between an algebraic specification and an algebra.

In the approach of 'algebraic semantics', the meaning of a specification is given by a class of algebras that is uniquely associated to the specification. In the current work on abstract data types there are two complementary semantics associated to algebraic specifications. To describe these we first need some additional concepts.

1.4 The Herbrand Construction

Recall that a specification consists of a pair (S,E) where S is a signature and E is a set of axioms. Let ALG(S) denote the set of all S-algebras, algebras whose signature is S. Given S, how do we know that there exists any algebras in Alg(S) ? And given that such S-algebras exist, how do we know that there exist S-algebras that satisfy the axioms E ?

Given a specification (S,E), define the set of all formal free terms, Term(X,S), according to the following rules:

1. If t is a 0-ary operator or free variable of sort s, then t is a term of sort s.

2. If t1, t2, ... ,tn are terms of sorts s1, s2, ...,sn, and t is an operator of characteristic

$$t:s1,s2,....sn \rightarrow s$$

then

$$t(t1,t2,...,tn)$$

is a term of sort s.

Let Term(S) denote the set of all terms that do not contain any free variables. Note that both Term(X,S) and Term(S) consist of terms of different sorts. Denote the terms in Term(S) of sort s by Term(S)(s). The sets Term(S)(s) can now be viewed as carriers in a S-algebra Term(S). The operations on this algebra are associated to the operator templates of S. If op is an operator template

of characteristic s1,s2,...,sn -> s define the operation f-op from Term(S)(s1),...,Term(S)(sn) to Term(S)(s) by:

f-op(t1,...,tn) = op(t1,...,tn)

where
t1,...,tn are terms of sort s1,...,sn

The formal construction used to create Term(S) is called the Herbrand construction in the mathematical literature.

In the case of the Boolean specification above, the term algebra consists of all the term expressions we can form abiding by the type characteristics of each operator template.

There is another equivalent description of the sets of terms determined by a signature. We can view the terms as strings on the alphabet consisting of the operator names, the comma, left and right parentheses, and some finite alphabet of symbols for free variables of different sorts. Then the set of terms forms a language on this alphabet with the following grammar:

For each sort s in S add the production rule:

<Term(S)> -> <Term(S)(s)>

For each operator of characteristic:

op: s1,s2,...,sn -> s

add the rule:

<Term(S)(s)> -> 'op('<Term(S)(s1)>','...','<Term(S)(sn)>')'

For each free variable X of sort s, add the rule:

<Term(S)(s)> -> 'X'

It is not difficult to see that the resulting grammar is LL(1), and therefore parsable by simple and efficient methods. In particular there are automatic parser generators that will take the signature of a specification as input and generate a table driven parser for terms defined for the given signature. The resulting parse tree can in fact be used as a representation of the term for use in rapid prototyping. Essentially this is the theoretical justification for the methods implemented in Guttag, Horowitz, and Musser [1978].

1.5

Congruences

An equivalence relation R on a S-algebra A is called a congruence if:

1. R-equivalent elements have the same sort

2. If (t1,t1'),(t2,t2'),...,(tn,tn') are pairs of R-equivalent elements of sorts s1,s2,...,sn and op is an operation of type s1,s2,...,sn -> s, then op(t1,...,tn) is R-equivalent to op(t1',...,tn').

If R is a congruence on a S-algebra A, then there is induced on the equivalence classes A/R a natural S-algebra, called the quotient algebra (Gratzer [1968]).

1.6 Initial Algebras

If (S,E) is a specification, there are two canonical congruences induced on the term algebra Term(S) by the axioms E. These two congruences then induce two quotient algebras on Term(S), called the 'initial quotient algebra' and 'final quotient algebra'.

The first congruence will be denoted Initial(S,E), and the second will be denoted Final(S,E). Initial(S,E) is defined by the following rule:

(t,t') are Initial(S,E)-equivalent
if and only if
t = t' can be proved from the axioms of E

The axioms as expressed are equations between terms of the 'free term algebra' associated to the algebra. This free algebra includes terms with free variables of the appropriate sort. For example, in the axioms for the Boolean type there occur axions such as:

And(True(),x) = x

The variable x is a free variable of sort Bool. The rules for proving equations from axioms and other proven equations are given by:

1. Any axiom is a proven equation. Any conditional axiom is a valid rule of inference for proving equations from proven equations.

2. If in a proven equation, every occurrence of a free variable is replaced by a single term of the same sort as the variable,

the resulting equation is proven.

3. If in an equation, some term is replaced by a term provably equal to it, the resulting equation is proven.

4. Any equation derived from proven equations by the use of the reflexive law, symmetric law, or transitive law for equality, is also proven.

With the above rules, it is not difficult to prove that the relation defined by all pairs of provably equal terms (with or without free variables) is a congruence.

1.7

Final Algebras

The set of terms of a specific sort s in the initial quotient algebra defined by a specification (S,E) is said to be 'trivial' if using the axioms E, any two terms t, t' of sort s are provably equal.

Given a specification (S,E), we say that an equation between two terms t = t' of a non-trivial sort s in the free term algebra is 'consistent' with the axioms E, if in the specification defined by the axioms E with the additional axiom t = t', the set of terms of sort s is not trivial. It follows, for example, that any provable equation is consistent. However, there may be consistent equations that are not provable. Define the relation Final(S,E) on terms of the free term algebra Term(S) by:

t, t' are Final(S,E) - equivalent
iff
t = t' is consistent with the axioms E

It can be proved that the relation Final(S,E) is a congruence. The corresponding quotient algebra is called the 'final algebra' defined by the specification (S,E).

1.8

Algebra Morphisms

So far, we have shown that a specification (S,E) determines two specific S-algebras, the initial S-algebra and final S-algebra. How are specifications related to other algebras ? For example, we have given a specification for the Boolean type. How is this specification related to an algebra that seems to realize the specification ?

Given two S-algebras A and B, a correspondence H associating each carrier set of A to a carrier set of B, and each operation of A to an operation of B is said to be an S-homomorphism if:

1. the correspondence between carrier sets preserves the sort type.

2. the correspondence between operators preserves the operator characteristics.

3. if t1,...,tn are elements of sorts s1,...,sn in A and op is an operator in A, then
H(op(t1,...,tn)) = H(op)(H(t1),...,H(tn)).

An S-homomorphism is called an S-monomorphism if each correspondence between carrier sets is injective. An S-homomorphism is called an S-epimorphism if each correspondence between carrier sets is a surjection. And an S-homomorphism is called an S-isomorphism if each correspondence between carrier sets is a bijection.

S-homomorphisms are intimately associated to S-congruences. If H is an S-homomorphism between S-algebras A and B, define the relation R(H) on the S-algebra A by:

For any sort s in the signature S, let A(s) be the carrier of sort s in the algebra A, and let H(s) be the correspondence determined by H between the carriers of A and B of sort s.

a, a' in A(s) are R(H) - equivalent
iff
H(s)(a) = H(s)(a')

It is not difficult to show that R(H) is an S-congruence. It is the S-congruence canonically associated to the S-homomorphism H.

Given any S-algebra A, there is a canonical S-homomorphism Val(A) from the term algebra Term(S) to A, determined by evaluating each formal term of Term(S) through its corresponding terms in A. There is then an S-congruence R(Val(A)) on Term(S) induced by Val(A). For convenience, this congruence will be denoted R(A).

1.9

Algebraic Semantics

At this point we have all the conceptual tools we need to precisely describe the two classes of algebras that will normally be used to interpret a specification (S,E). Each class of algebras will determine one of the two meanings that a specification determines, hence the terminology 'algebraic semantics'.

The 'initial algebra' semantics of a specification (S,E) is the class of all S-algebras A such that:

1. the S-homomorphism Val(A) from Term(S) to A is an S-epimorphism,

2. the S-congruence R(A) on Term(S) is identical with Initial(S,E).

The 'final algebra' semantics of a specification (S,E) is the class of all S-algebras A such that:

1. the S-homomorphism Val(A) from Term(S) to A is an S-epimorphism,

2. the S-congruence R(A) on Term(S) is identical with Final(S,E).

It follows that the initial quotient algebra is an element of the class of initial algebras, and similarly, the final quotient algebra is in the class of final algebras. It is not difficult to prove that any two S-algebras in the class of 'initial algebras' are S-isomorphic. Similarly all the S-algebras in the class of 'final algebras' are S-isomorphic. Thus, in effect, any algebra in the class of initial algebras is isomorphic to the quotient initial algebra constructed from the term algebra and the Initial(S,E) congruence. Similarly for final algebras.

The above characterizations of the meaning of a specification can be used to effectively determine if a specification 'means' what we want it to mean. For example, in the example for Boolean, we started with an algebra consisting of a set {T,F} and operations '&' and '¬'. The operations were defined explicitly with a truth table. We then created a formal specification Boolean that we claimed captured the 'essence' of the boolean data type. Specifically we should be able to prove that the explicit algebra is a member of the class of initial or final algebras defined by the specification. Define a S-homomorphism H from Term(Boolean) to algebra A by:

```
True()   -> T
False()  -> F
Not      -> ¬
And      -> &
```

H is clearly a surjection from Term(Boolean) to {T,F}. Consider the relation R(A) induced on Term(Boolean) by A. We will show that it equals the relation Initial(S,E), where E is the set of axioms. We have to show that two terms evaluate in A to the same value if and only if the terms are provably equal using the axioms in E. Let the size of a term be the number of operators in the term. We will first prove by induction on the size n of the largest of the two terms that if two terms evaluate in A to the same result, then they are provably equal from E.

First we prove the following:

Lemma: if t evaluates to T, then t is provably equal to True, and if t evaluates to F, then t is provably equal to False.

Proof: Induct on the size n of t. If n = 1, the result is obvious. Assume true for n <= N and assume n = N + 1. The term t has the form t = Not(x) or And(x,y) for some terms x and y of size <= N. If t = Not(x), and t evaluates to T, then Not(Not(x)) = x evaluates to ¬T = F, and x has size <= N , therefore x is provably equal to False. But then t = Not(x) is provably equal to Not(False), hence provably equal to Not(Not(True)) = True. Similarly if t evaluates to F. If t = And(x,y), and t evaluates to T, then x and y must evaluate to T also. Since the size of x and y <= N, x and y are provably equal to True. But then from the axiom And(True,x) = x, it follows that t = And(x,y) is provably equal to True. Similarly if t evaluates to F.

Clearly every expression evaluates to T or F. Therefore if two expressions evaluate to the same value by the above they are provably equal to each other. Conversely, if two expressions are provably equal to each other, then they must evaluate to the same value, since the operators corresponding to each operator template satisfies the corresponding axioms.

Thus the quotient algebra defined by Term(Boolean)/Initial(S,E) is isomorphic to the algebra A, and A is an 'initial algebra' for the specification.

Moreover, note that the 'trivial' algebra consisting of the {a} and the operations:

$$T = F = a, \quad \neg(a) = a, \quad \&(a,a) = a$$

has the correct signature but is not in the class of 'initial algebras' defined by the specification, since in this algebra True and False evaluate to the same value, but are not provaoly equal.

Is the algebra A in the class of 'final algebras' defined by the same specification ? Equivalently, are the relations R(A) and Final(S,E) the same ? By what we have already proved it is sufficient to ask whether there are terms t, t' that are consistent, but not provably equal. If t and t' are consistent, then from E and the equation t = t', it does not follow that True and False are provably equal. But if t and t' are not provably equal, then they evaluate to different values T and F, say. But then by the lemma t is provably equal to True and t' is provably equal to False, and hence from the equation t = t', True is provably equal to False. Therefore if t and t' are consistent, they are provably equal. Clearly if t and t' are provably equal, then they are consistent.

The above arguments show that for the specification Boolean, the initial algebra and final algebra semantics are the same. It is a theoretical fact that there exist specifications for which this is not the case. This will become fairly evident when the computability of a specification is discussed in a later section.

In the case of Boolean, the axioms enable us to formally evaluate each term in terms of constant terms. For example, to evaluate And(Not(False),True):

```
And(Not(False),True) = And(True,Not(False))
And(True,Not(False)) = Not(False)
Not(False) = Not(Not(True))
Not(Not(True)) = True
```

In fact, the tables describing the operations explicitly (which also determine an equational specification, without free variables) is derivable from the above equations. The advantage of the equational specification with free variables is its compactness.

Ordinarily, it cannot be assumed that every term is provably equal to a constant term. It is often the case in the theory of abstract data types for example, that the number of distinct classes of unequal terms is infinite.

2.0

Properties of Specifications

Now that the basic concepts of our methodology have been described, we want to determine if it can be used to fulfill the requirements of a methodology for specifications.

2.1

Equivalence of Specifications

First it is possible to define when two specifications with the same signature have the same meaning, in the two interpretations of meaning discussed above. Two specifications (S,E) and (S,E') with the same signature have the same initial algebra semantics when they determine the same class of initial algebras. Similarly, they have the same final algebra semantics when they determine the same class of final algebras. It follows from the above that two specifications (S,E) and (S,E') have the same meaning in initial algebra semantics if and only if the axioms E are provably equal from the axioms E' and conversely. Similarly two specifications (S,E) and (S,E') have the same final algebra semantics if and only if any equation of formal terms t = t' is consistent with E if and only if it is consistent with E'.

The above definitions allow us to determine when two specifications with the same signature but different axioms have the same meaning. What about specifications that have different signatures ?

Assume that (S,E) and (S',E') are two specifications. A correspondence H between S and S' is called a Poincare transformation if:

1. for every sort s in S, H(s) is a sort in S'

2. for every operator op in S of characteristic
   s1,s2,...,sn -> s,
   H(op) is an operator in S' of characteristic
   H(s1),H(s2),...,H(sn) -> H(S)

Clearly, H induces a correspondence between Term(S) and Term(S'). If H is bijective between S and S', then H induces a bijection between Term(S) and Term(S'). Thus each equation t = t' in Term(S)is mapped to an equation H(t) = H(t') in Term(S'). In this case the specifications (S,E) and (S',E') are semantically equivalent in the sense of initial algebras, or final algebras, when the specifications (S',E') and (S',H(E)) are, and the question of equivalence reduces to the previous case. These cases cover the cases that correspond to a renaming of sorts or operators, in addition to a change in the axioms.

In general, two specifications have the same semantics when they determine the same class of algebras.

2.2

The Adequacy and Computability of Algebraic Specifications

In this section we examine the ability of this specification methodology to define all the types of objects that we might want to define. This is the 'adequacy' problem. Bergstra and Tucker [1983] have written a series of papers dealing with this question and we will summarize their ideas here.

Given a specification (S,E) and an S-algebra A, we say that A is 'effectively presented' whenever we possess an effective enumeration of its elements and we can effectively calculate its operations. The algebra A is said to be a 'semicomputable algebra', or a 'cosemicomputable algebra' if in addition the equality relation of A is recursively enumerable, or co-recursively enumerable, respectively. (Recall that a set is co-recursively enumerable if its complement is recursively enumerable). A is a 'computable algebra' when equality is decidable.

Since for initial semantics two terms are equal if and only if they are provably equal from the axioms, and since all such proofs can be enumerated, the initial algebra of a specification is semicomputable. It is less obvious, but also true, that the final algebra of a specification is co-semicomputable. Note that if an algebra is both semicomputable and co-semicomputable, then the equality relation between terms is decidable.

Of more interest to the question of adequacy ,is the converse of these facts. Is every semicomputable algebra the initial algebra of some specification ? Is every co-semicomputable algebra the final algebra of some specification ? Is every computable algebra both the final algebra of some specification and the initial algebra of some specification ? Bergstra and Tucker [1983] have been able to prove the second and last of these assertions, given that the specifications may include conditional equations.

Theorem (Bergstra and Tucker). An algebra is semicomputable if and only if it is the final algebra of a finite conditional specification.

Theorem (Bergstra and Tucker). An algebra is computable if and only if it is both the initial algebra of a finite conditional specification and the final algebra of a finite conditional specification.

Bergstra and Tucker also show that the set of functions computed by LOOP programs on the natural numbers compose a data type which has a finite conditional specification using final algebra semantics, but does not possess an effective specification of any kind using initial algebra semantics.

The fact that individually, initial and final semantics do not attach the same semantics to algebraic secifications is one reason why both concepts need to be introduced. There is another more practical reason however. It is reasonable to require that the specifications of some resources be computable. Presumably the functional behavior of most physical devices is decidable.

Unfortunately, the characterization of the computability of a specification given by Bergstra and Tucker is not generally practical. In order to make the methodology practical we need simple but broad criteria for insuring the computability of a specification.

One approach to this problem is to determine if the axioms define rewrite rules that allow one to prove that each term without free variables can be reduced to a normal form. To do this it is necessary to prove some kind of Church Rosser property for normal forms, to guarantee their uniqueness.

2.3 Specification Syntax

Now that the basic concepts underlying the methodology of algebraic specifications have been introduced we can illustrate how these concepts are used in practice. First we need to establish a syntax for describing a specification. Rather than give a grammar, we will describe the templates used for specifications. An example of the form of a specifications was given above for the boolean data type. A template for this form is:

SPECIFICATION <Specification-identifier>

OPERANDS
<Operand-list>

OPERATORS
<Operator-list>

AXIOMS
<Axiom-list>

It is convenient to have a specification syntax that facilitates the combination and extension of specifications to provide a modular approach to complex specifications. Although ultimately, a specification should always be expressible in the above form, it is convenient to eliminate expressive redundancy through a more complex syntax. For example, we want to provide a facility for readily incorporating a commonly used specification into a new

specification. The syntax used to incorporate a previously defined specification into a new specification is:

```
SPECIFICATION <New-specification-identifier>

        EXTEND
                <Old-specification-identifier-list>

        BY

        OPERANDS
                <Operand-list>

        OPERATORS
                <Operator-list>

        AXIOMS
                <Axiom-list>
```

The semantics of such an extended specification are the semantics of the composite specification as if it were written without extension. The properties of the extension may involve the operands and operators of both specifications. Thus in an extension the semantics of a previously defined specification could change within the context of new operands, operators, and properties. In most cases such semantic changes are undesirable. Thus, we need a criterion for when such changes do not occur. In the process of determining such a criterion we will also illustrate the use of the semantic interpretation associated to a specification.

Assume (S,E) and (S',E') are two specifications for which S is a subset of S' and E is a subset of E'. First let us consider initial algebra semantics. In this case, the relation Initial(S',E') is a subset of the relation Initial(S',E') since every formal term in Term(S) is also a term is Term(S'), and if t and t' are provably equal in E, they are provably equal in E' since E is a subset of E'. The terms in Term(S), viewed as terms in Term(S') will have the same properties in (S',E') as the properties in (S,E) if and only if any two terms t and t' in Term(S) that are Initial(S',E') related are also Initial(S,E) related. In other words, if the restriction of the relation Initial(S',E') to Term(S) equals the relation Initial(S,E).

We say that a specification (S,E) is persistent in an extended specification (S'E'), in the sense of initial semantics, if the relation Initial(S',E') restricted to Term(S) equals the relation Initial(S,E).

If S is a 'subsignature' of S' and A and A' are S-algebras and S'-algebras respectively, then A is said to be a subalgebra of A' if there is a S-monomorphism from A to A'. Similarly, we say that a specification (S,E) is

persistent in an extended specification (S',E'), in the sense of final semantics, if the relation Final(S',E') restricted to Term(S) equals the relation Final(S,E).

The following theorems can now be proved.

Theorem: (S,E) is a persistent subspecification of (S',E'), in the sense of initial semantics, if and only if the initial algebra of (S,E) is a subalgebra of the initial algebra of (S',E').

Theorem: (S,E) is a persistent subspecification of (S',E'), in the sense of final semantics if and only if the final algebra of (S,E) is a subalgebra of the final algebra of (S',E').

Although these results are essentially theoretical they are the basis for at least one practical test for persistency. In the example given for boolean we showed how to verify the correctness of a specification by establishing a mapping between the specification and an algebra that is a canonical representative of the data type. Specifications may often be obtained in this way in practice. If the data type is computable, this algebra is both a final and initial algebra for the specification. If the specification is extended, then it will be persistent in the extended specification if and only if its canonical algebra is a subalgebra of a canonical algebra for the extension.

2.4

Derived Algebras and Specifications

Given an algebra of signature S and a term t(x1,x2,...,xn) with free variables of sorts s1,s2,...,sn and return sort s, we can view t as defining an operator of characteristic s1,...,sn -> s. Such an operator is called a derived operator of A. More generally, an algebra B, whose sorts are a subset of the sorts of A, and whose operators are derived operators of A, is called an algebra derived from A. The signature of B is also said to be derived from the signature of A. For example,

Imp(x1,x2) = Not(And(x1,Not(x2))

is a derived operator in Boolean. This is the well known 'implies' operator.

If (S,E) and (S',E') are specifications, and S' is a signature derived from S, we say that the initial semantics of (S',E') are consistent with the initial semantics of (S,E) if given any terms t and t' of Term(S'), if (t,t') is in Initial(S',E') then (t,t') is in Initial(S,E). We say

that the initial semantics of (S',E') are faithful to the initial semantics of (S,E), if for any t, t' in Term(S'), (t,t') is in Initial(S,E) if and only if (t,t') is in Initial(S',E'). Similarly for final semantics.

2.5

Interpretations and Implementations

The practical problem we are attempting to solve involves software portability. Specifically, we want to be able to specify resource interfaces in an implementation independent manner beginning with physical interfaces up to problem solving interfaces. In this view, we want to successively build layers of resources implemented on previously defined sublayers. For this reason it is natural to expect that an 'implementation' must be determined by relating one specification to another.

Alternatively, we want to realize a specification in terms of resources for which we do not have specifications. We call such a realization an interpretation. An interpretation is specified by associating to each sort a description of the values that will realize the sort, and to each operator, a function procedure operating on the appropriate values. An interpretation is valid if it defines an algebra in the semantic class chosen for the specification. Thus an interpretation associates the specification to a specific 'algebra', that is realized by a set of function procedures and data types in a program.

To implement a specification A in terms of another specification B ought to mean that the sorts, operators, and properties of A ought to be implementable from those of B. Therefore we make the following definition:

Given specifications (S,E) and (S',E'), an implementation of (S,E) on (S',E'), in the sense of initial semantics, is a Poincare transformation H from the signature S to a derived signature S'' of S' with the property that the homomorphism induced from Term(S) to Term(S'') is consistent with the congruences Initial(S,E) on Term(S) and Initial(S',E') on Term(S''). Or equivalently, for t,t' in Term(S),

if (t,t') is in Initial(S,E)
then
(H(t),H(t')) is in Initial(S',E')

An implementation H is faithful, in the sense of initial semantics, if in fact it satisfies:

(t,t') is in Initial(S,E)

if and only if
(H(t),H(t')) is in Initial(S',E')

We clearly have similar definitions for implementations in the sense of final semantics.

Theorem: There is an implementation of (S,E) on (S',E') in the sense of initial (final) semantics if and only if there is a homomorphism from the initial (final) algebra of (S,E) to a derived algebra of the initial (final) algebra of (S',E').

And,

Theorem: There is a faithful implementation of (S,E) on (S',E') in the sense of initial (final) semantics if and only if there is a monomorphism from the initial (final) algebra of (S,E) to a derived algebra of the initial (final) algebra of (S',E').

Though these theoretical results seem comforting, are there any practical methods for determining if a correspondence is an implementation or if such an implementation is faithful ?

If H is a correspondence from (S,E) to (S',E') that assigns to each sort s in S, a sort H(s) in S', and to each operator f in S, a derived operator H(f) of S', then H is an implementation in the sense of initial semantics if and only if for every axiom t = t' in E, the equation H(t) = H(t') is provably equal from E'.

2.6

Parameterized Specifications

Certain kinds of resources are naturally parameterized. For example, in the case of the string data type, strings of integers, or strings of characters, or strings of bits all require many of the same operations and share many of the same properties, yet in most cases must be defined separately. Parameterized specifications are specifications used to specify resources that may be instantiated for a number of different resource parameters, but are not uniquely associated to any of them. In this section, we introduce the basic ideas with a minimum of discussion. For details of some of the theoretical work in this area see Ehrich [1982].

3.0

Parameterized Specification Syntax.

The template for a parameterized specification has the form:

```
SPECIFICATION <specification-identifier>

        PARAMETERS
                OPERANDS
                        <parameter-operand-list>

                OPERATORS
                        <parameter-operator-list>

                AXIOMS
                        <parameter-axiom-list>

        OPERANDS
                <operand-list>

        OPERATORS
                <operator-list>

        AXIOMS
                <axiom-list>
```

The axioms for the body of the parameterized specification may include operand classes and operators of the parameter part, in addition to the other operand classes and operators. The axioms of the parameter part may include only operand classes and operators of the parameter part. The parameter part describes the operand classes, operators, and axioms that must be specified in any invocation of the parameterized specification. The syntax of an invocation is:

```
        . . .
        <parm-spec-ident> ( <spec-ident> )
                WHERE
                        OPERANDS
                          <spec-operand> IS <parm-spec-operand>
                          . . .

                        OPERATORS
                          <spec-operator> IS <parm-spec-operator>
                          . . .
```

Thus in an invocation, a correspondence is established between operands and operators of one specification (actual parameters) and the operand and operators of the parameter part of the parameterized specification. For semantics, the specification resulting from such an invocation is viewed as an extension of the specification that supplies the actual

parameters. To be a valid invocation, the actual parameters must satisfy the parameter axioms as a consequence of the axioms that they already satisfy. Also, to be a valid invocation, the specification supplying the parameters must be persistent in the resulting specification. In the resulting specification, as in an extension of a specification, the semantics must be the same as in the actual parameter subspecification. The significant point from the practical viewpoint is that in this case, as well as in the case of speicification extension, we need practical criteria for determining the persistence of a subspecification.

4.0

Conclusions

Current practical approaches to functional specification are not based on any rigorous foundation, or developed in the context of any general theory. This fact precludes the development of portable logical interfaces, or the sytematic specification of a hierarchy of logical interfaces. The approach taken here attempts to resolve some of these problems. It begins with a model close to practice, the idea of an 'algebra', consisting of operators and operands, and establishes a syntax and semantics for specifications. Transformations between specifications at the syntactical level are described by Poincare transformations, a concept used for a long time in logic to describe the same process of establishing a correspondence between the names of the entities of one formal theory with those of another. The semantics are established through interpretations of the syntactical elements as elements of certain algebras, depending on the semantics chosen. Relations between the semantics of two specifications are established by homomorphisms between their respective interpretations. Every important concept associated to specifications can now be given rigorous definitions.

At the practical level, concrete resources, either in software or hardware, are viewed as defining concrete algebras that may serve to interpret a specification faithfully. Whether this assumption is reasonable remains to be seen. This is the most obvious question open to further research. Moreover, it seems apparent that much of the theorem proving technology developed in recent years may find an application to the analysis of specifications. In any case, this approach has served to form a rigorous foundation for a theory of specification, and to expose some of the difficult issues that must be addressed by any approach.

5.0

A Sample Abstract Machine

To illustrate the practical potential of this specification technique, a specification of an abstract processor is included below. The first part defines the data types required to define memory, values, and states. The later part defines the operations and instructions of the processor. The basic idea for such a machine can be found in Fasel[1980].

The specifications below combine to specify a sample abstract machine. Metasymbols describing the form of the specification are capitalized.

```
CONVENTION
        A binary op X: Elem, Elem -> Elem is
        COMMUTATIVE
                if X(x,y) = X(y,x)
        ASSOCIATIVE
                if X(X(x,y),z) = X(x,X(y,z))

SPEC Boolean
        SORTS
                Bool
        OPS
                True:   -> Bool
                False:  -> Bool
                Not:    Bool -> Bool
                And:    Bool,Bool -> Bool
        AXIOMS
                Not(True) = False
                Not(Not(x)) = x
                And(True,x) = x
                And(False,x) = False
                And is COMMUTATIVE
                And is ASSOCIATIVE

SPEC Natural

        SORTS
                Nat
        OPS
                0: -> Nat
                Next: Nat -> Nat


SPEC Integer

        EXTEND  Boolean,
                Natural
        WITH

        SORTS
```

```
                Int
        OPS
                0: -> Int
                Next: Int -> Int
                Neg: Int -> Int
                Add: Int,Int -> Int
                Sub: Int,Int -> Int
                Lte: Int,Int -> Bool
                Abs: Int -> Nat

        NOTATION
                -x is Neg(x)
                x+y is Add(x,y)
                x-y is Sub(x,y)
                x<=y is Lte(x,y)

        AXIOMS
                Add is COMMUTATIVE, ASSOCIATIVE
                Add(x,0) = x
                Add(x,Next(y)) = Next(Add(x,y))
                Neg(0) = 0
                Next(Neg(Next(x))) = Neg(x)
                Add(x,Neg(x)) = 0
                Sub(x,y) = Add(x,Neg(y))
                Lte(x,x) = True
                Lte(x,y) => Lte(Next(x),y)
                Lte(x,y) = Lte(Sub(x,y),0)

        END

SPEC Character

        SORTS
                Char
        OPS
                A: -> Char
                a: -> Char
                B: -> Char
                b: -> Char
                . . .

        END

SPEC Identifier

        EXTEND  Boolean WITH

        SORTS
                Id
        OPS
                Register: -> Id
                Main: -> Id
                Disk: -> Id
                Display: -> Id
                Eqid: Id,Id -> Bool
```

```
        AXIOMS
                Eqid(x,x) = True

        END


SPEC String

        PARAMETER

                SORTS
                        Elem
        EXTEND
                Natural
        WITH
        SORTS
                Str
        OPS
                Null: -> Str
                Make: Elem  -> Str
                Cat: Str,Str -> Str
                Len: Str -> Nat
                Head: Str -> Elem
                Tail: Str -> Str
        AXIOMS

                Len(Null) = Natzero,
                Len(Make(a)) = Next(Natzero)
                Head(Make(a)) = a
                Tail(Null) = Null
                Tail(Make(a)) = Null
                Cat is ASSOCIATIVE
                Cat(s,Null) = Cat(Null,s)
                Cat(s,Null) = s
                Head(Cat(Make(a),s) = a
                Tail(Cat(Make(a),s) = s
                Len(Cat(Make(a),s)) = Next(Len(s))
                Len(Cat(s,Make(a)) = Next(Len(s))

        END

SPEC Bitstring

        String(Boolean)

        WHERE
                Elem IS Bool
        END

SPEC Chrstring

        String(Character)

        WHERE
                Elem IS Char
```

```
        END

SPEC Data-values

        EXTEND
                Boolean
                Natural
                Integer
                Character
                Bitstring
                Chrstring

        WITH
        SORTS
                Val
        OPS
                Errval: -> Val


                Val-to-bool: Val -> Bool
                Val-to-nat: Val -> Nat
                Val-to-int: Val -> Int
                Val-to-chr: Val -> Chr
                Val-to-bitstr: Val -> Bitstr
                Val-to-chrstr: Val -> Chrstr

                Bool-to-val: Bool -> Val
                Nat-to-val: Nat -> Val
                Int-to-val: Int -> Val
                Charval: Char -> Val
                Bitstr-to-val: Bitstring -> Val
                Chrstr-to-val: Chrstring -> Val

        AXIOMS
                FOR X = Bool,Nat,Int,Char,Bitstr,Chrstr

                        Val-to-X(X-to-val(x)) = x

                        X-to-val(Val-to-X(v)) = v

                FOR X,Y = Bool,Nat,Int,Char,Bitstr,Chrstr
                X ~= Y,

                        X-to-val(Val-to-Y(v)) = Errval


SPEC Addresses

        EXTEND
                Identifier,
                Boolean
        WITH
        SORTS
                Addr
```

```
        OPS
                Startaddr: Id -> Addr
                Nextaddr: Addr -> Addr
                Equaladdr: Addr, Addr -> Bool

        AXIOMS
                Equaladdr is an EQUIVALENCE
                Equaladdr(Startaddr(x), Startaddr(y)) = Equalid(x,y)
                Equaladdr(Nextaddr(x), Nextaddr(y)) = Equaladdr(x,y)


SPEC Operators

        EXTEND
                Data+values
        WITH
        SORTS
                Monop,
                Binop,
                Relop
        OPS
                Boolnot: -> Monop
                Booland: -> Binop

                Natadd:  -> Binop

                Intadd:  -> Binop

                Chrstrcat: -> Binop

                Bitstrcat:  -> Binop

                Intgt:  -> Relop

                . . .

                Applymonop: Monop,Val -> Val
                Applybinop: Binop,Val,Val -> Val
                Applyrelop: Relop,Val,Val -> Val

        AXIOMS
                Applymonop(Boolnot,v) =
                        Boolval(Not(Val+to+bool(v)))
                Applybinop(Booland,v1,v2) =
                        Boolval(And(Val+to+bool(v1),Val+to+bool(v2)))

                ... etc.

SPEC Instructions

        EXTEND
                Operators
        WITH
```

```
        SORTS
                Instr
        OPS
                Monad: Monop,Addr,Addr -> Instr
                Binad: Binop,Addr,Addr,Addr -> Instr
                Mov: Addr,Addr -> Instr
                Movi: Addr, Val -> Instr
                Jmp: Addr -> Instr
                If: Relop,Addr,Addr,Addr  -> Instr
                Push: Addr, Stk -> Instr
                Pop: Addr, Stk -> Instr
                Call: Addr, Stk -> Instr
                Ret: Stk -> Instr
                Halt: -> Instr

SPEC Values

        EXTEND
                Data+values,
                Instructions
        WITH
        OPS
                Instr+to+val: Instr -> Val
                Val+to+instr: Val -> Instr
        AXIOMS
                Val+to+instr(Instr+to+val(i)) = i
                Instr+to+val(Val+to+instr(v)) = v

                FOR X = Bool,Nat,Int,Chr,Bitstr,Chrstr

                        Instr+to+val(Val+to+X(v)) = Errval


SPEC Machinestate

        EXTEND
                Values,
                Instructions
        WITH
        SORTS
                State
        OPS
                Initialstate: -> State
                Store: Val,Addr,State -> State
                Fetch: Addr, State -> Val

        AXIOMS
                Fetch(a,Initialstate) = Errval
                Fetch(a,Store(v,a,s)) = v
                Store(Fetch(a,s),a,s) = s

SPEC Machine

        EXTEND
                Machinestate
```

```
WITH
OPS
        Program: Addr,State -> State
        Execute: Instr,Addr,State -> State
AXIOMS
        Program(a,s) =
                Execute(Val-to-instr(Fetch(a,s)),a,s)
        Execute(Mov(a1,a2),a,s) =
                Program(Next(a),Store((Fetch(a1,s),a2,s))
        Execute(Movi(a1,v),a,s) =
                Program(Next(a),Store(v,a1,s))
        Execute(Jmp(a1),a,s) =
                Program(a1,s)
        Execute(If(r,a1,a2,b),a,s) = Program((Cond(Val-to-bool
        (Applyrelop(r,Fetch(a1,s),Fetch(a2,s))),b,Next(a)),s)
        Execute(Halt,a,s) = s
        Execute(Monad(m,a1,a2),a,s)) = Program(Next(a),
        Store(Applymonop(m,Fetch(a1,s)),a2,s))
        Execute(Binad(b,a1,a2,a3),a,s) = Program(Next(a),
        Store(Applybinop(b,Fetch(a1,s),Fetch(a2,s)),a3,s))
```

Acknowledgements

This research was supported at the Naval Postgraduate School under the Foundation Research Program.

References.

A. Bergstra and J.V. Tucker, A natural data type with a finite equational final semantics specification but no effective equational initial semantics specification. Bull EATCS, 11 (1980), pp. 23-33.

A.Bergstra and J.V. Tucker, Initial and Final Algebra Semantics for Data Type Specifications: Two Characterization Theorems. SIAM J. Comput. Vol. 12, No.2, May 1983.

Bundy, Alan, "The Computer Modelling of Mathematical Reasoning", Academic Press, New York, 1983.

Ehrich, H.D., On the Theory of Specification, Implementation, and Parametrization of Abstract Data Types, J.of ACM. 29, No.1, Jan.,1982.

Fasel, Joseph, "Programming Languages as Abstract Data Types - Definition and Implementation ", Ph.D. Thesis, Purdue University August, 1980.

Gratzer.G., Universal Algebra, D.Van Nostrand, New York, 1968.

J.A. Goguen, J.W. Thatcher,E.G. Wagner and J.B. Wright, An initial algebra approach to the specification, correctness, and implementation of abstract data types. Current Trends in Programming Methodology IV, Data Structuring, R.T.Yeh,ed.,Prentice-Hall, Englewood Cliffs NJ,1978,pp.80-149.

Guttag, J.V., Horowitz, E.,and Musser, D.R., "Abstract Data Types and Software Validation", pp. 1048-64, Comm. ACM., V.21, No.12, Dec. 1978.

Hoffman,C.M., O'Donnell,M.J.,"Programming with Equations", pp.83-112, ACM Trans. on Prog. Lang., Vol.4, No.1, January,1982

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	3
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93943	1
Chariman, Code 52ML Department of Computer Science Naval Postgraduate School Monterey, CA 93943	20
Associate Professor Daniel L. Davis, Code 52Vv Department of Computer Science Naval Postgraduate School Monterey, CA 93943	30